Modular Components for Network Address Translation

Eddie Kohler
ICSI Center for Internet Research
kohler@icir.org

Robert Morris
MIT Lab for Computer Science
rtm@lcs.mit.edu

Massimiliano Poletto
Mazu Networks
maxp@mazunetworks.com

Abstract

We present a general-purpose toolkit for network address translation in Click, a modular, component-based networking system. Network address translation, or NAT, was designed to allow disparate address realms to communicate. The components of our toolkit can be combined in a variety of ways to implement this task and many others, including some that, superficially, have nothing to do with address translation. Our NAT components are more flexible than monolithic alternatives. They concern themselves solely with address translation; separate components handle related functions, such as classification. The user can choose where network address translation takes place in relation to other router functions; combine multiple translators in a single configuration; and use NAT in unintended, surprising ways.

We describe our design approach, demonstrate its flexibility by presenting a range of examples of its use, and evaluate its performance. Our components have been in use in a production environment for over eighteen months.

1 Introduction

This paper presents a general-purpose toolkit for network address translation (NAT), or, more generally, for rewriting packets' addresses and port numbers. The toolkit is implemented as a family of *elements* in the Click modular router [6, 12]. Individually, each element is not necessarily more powerful than other NATs or traffic redirectors. However, the ability to combine elements into new arrangements makes our system more flexible than any single monolithic system. Our NAT demonstrates the advantages of flexible, component-based networking systems, and Click in particular. It also provides a case study of how to design Click components for a relatively complex networking task.

The rewriting elements divide into three categories: *rewriters*, which store NAT state and modify packets; *mapping plugins*, which implement arbitrarily complex policies for assigning new addresses to packets; and *application-level gateways*, which help protocols pass through the NAT. Each individual element has clear, specified semantics, so a given configuration is easily analyzed and understood. The elements may be combined in arbitrary ways to implement address translation and tasks that, superficially, have nothing to do with address translation.

Conventional NATs, such as those distributed with Linux and *BSD, implement common rewriting tasks well, but they are not flexible enough to handle unusual situations, such as those requiring multiple rewriters, and it may be difficult to tell how they will behave in corner cases. Click performs on par with these conventional NATs while remaining flexible and understandable.

The contributions of this paper are a description of Click's modular, component-based NAT implementation; novel uses of NAT facilitated by Click's NAT components; and, more generally, an application of Click's component design principles, showing how those principles lead to flexible designs.

The next section gives some background on network address translation. Section 3 describes the design and implementation of the family of rewriting elements. Section 4 presents a variety of examples of how the rewriter can be used, and Section 5 analyzes its performance. Sections 6 and 7 discuss related work and conclude. Finally, Appendix A provides a whirlwind overview of the Click system.

2 Network address translation

This section describes network address translation in general, then lists the features desirable in a modern NAT.

2.1 Overview

Packets contain both addressing information, such as IP addresses and TCP/UDP port numbers, and data. Network address translation (NAT) works from the simple insight that addressing information is largely independent of data. For many Internet protocols, IP addresses and TCP port numbers appear only in packet headers; the data transferred between endpoints is independent of address and port. Thus, a middle box could change the addressing information on

Much of this research was performed while all three authors were at MIT LCS. It was supported there by a National Science Foundation (NSF) Young Investigator Award and the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory under agreement number F30602-97-2-0288. In addition, Eddie Kohler was supported by a National Science Foundation Graduate Research Fellowship.

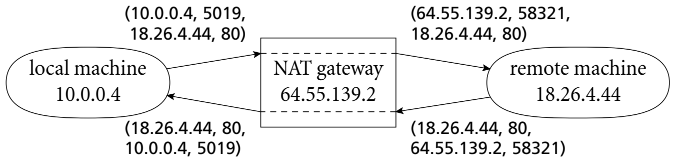

Figure 1—Network address port translation. A packet is represented by its flow identifier, the quadruple (source address, source port, destination address, destination port).

passing packets without affecting the semantics of the end-to-end connection, or requiring changes at end hosts.

Basic NAT was introduced in the early nineties to reduce the pressure for globally allocated IP addresses [9]. With NAT, an organization could assign unique, private IP addresses to each of its hosts, reserving just a few globally-allocated addresses to be shared among the hosts. All packets leaving or entering the organization would pass through a NAT box. When an internal host sent a packet to the Internet, the NAT box would temporarily assign that host a global IP address, and rewrite its packets to use that address. Reply packets from the Internet also need rewriting, since the internal computers recognize their private IP addresses, not addresses from the global pool. The NAT box reclaims global addresses after some period of inactivity. Note that only the internal hosts with temporarily-assigned global addresses are reachable from the Internet. A newer, and now more common, variant, NAPT (network address port translation) [17], rewrites both IP addresses and TCP/UDP port numbers. The rewriting unit is a TCP or UDP connection, not an IP address. This lets multiple hosts share the same global address, further reducing the number of global addresses required, but it generally prevents externally-initiated connections. Figure 1 illustrates NAPT in action.

NAT's other uses include network security (preventing externally-initiated connections), transparently load-balancing requests among servers [16], and allowing disparate address realms to communicate, including realms using different IP versions [13, 18]. Other uses have been proposed, such as creating network redundancy [11].

NAT is sometimes seen as a blemish on the Internet architecture [10], but it deserves its place in the network administrator's toolbox. Furthermore, its underlying technology—that is, the ability to keep track of active addresses and/or connections—can be useful for other network tasks, such as building transparent proxies, when the NAT implementation is flexible enough.

2.2 Implementation characteristics

This section lists the important properties of modern NAT implementations.

- The **addressing unit** used to look up a packet. For Basic NAT, this is a single internal IP address. For NAPT, it is a *flow identifier*: the tuple of source address, destination address, source port, destination port, and IP protocol (TCP or UDP). Other forms of NAT might choose other units.

- **Mapping table**. Every NAT includes a table recording the current mappings between internal and external addressing units. The table must allow translation in both directions—for packets leaving the internal network, which generally have their source addresses changed, and for packets entering that network, which generally have their destinations changed.

When a packet arrives, the NAT will extract its addressing unit and look that up in the mapping table. If a mapping is present, the NAT rewrites the packet according to the mapping and sends it on its way. If no mapping is present, the packet is called *fresh*, and is handled according to the NAT's *fresh packet rules*.

- **Fresh packet rules**. Usually, a fresh packet causes a new mapping to be installed in the table; the NAT then rewrites the packet according to that mapping and sends it on its way. The NAT must allow the user a fully general choice of address and port ranges used to construct the new mapping. NATs also support different actions for different classes of packet. For example, NAPT drops the class of fresh packets that originated on the external network, rather than creating new mappings. However, a NAPT configuration might pass some externally-initiated packets through to internal servers, such as rewriting all Web requests to head to an internal Web server.

- **Garbage collection**. The NAT should garbage-collect mappings after some period of inactivity, making their external addressing units available for reuse. This policy generally involves timeouts. Most TCP connections on a NAPT are garbage-collected soon after FIN flags indicate the connection has closed. A flexible NAT may wish to provide more control over state, by limiting the rate at which new mappings can be introduced, for example.

- **APIs for "application-level gateways" (ALGs)**. Some protocols mix addressing information in the data stream. To support these protocols, the NAT must export APIs that allow application-specific translation agents, also known as ALGs [17], access to address mappings. The classic example is FTP [14], which opens a new data connection for each file transfer. The FTP control connection gives, in ASCII, the address and port to be used for the data. An FTP gateway examines and occasionally modifies FTP control packets to change the embedded addresses and ports. It also installs a new NAT mapping for each data connection.* Changing the

*In normal usage, the server contacts the client to open a data con-

ASCII IP addresses and ports may alter the packet's length, which requires TCP sequence number adjustments.

- **Flexible placement** relative to other network processing tasks. The machine running NAT may be conceptually "inside" or "outside" the NAT's boundary; that is, its services may have a public or private address. NATs should support either placement.

2.3 Existing NAT implementations

NAT is well supported by many of today's routers and operating systems. Here, we describe Linux 2.4's Netfilter-based NAT [1] as a typical implementation. We also describe Netfilter NAT's limitations. Its single mapping table and limited placement relative to other networking tasks limit its usefulness for advanced and unexpected NAT configurations. Section 6 describes other commonly deployed NATs; they suffer from the same limitations.

The Netfilter facility examines packets at each of a fixed set of points in the Linux IP forwarding path: on entry from interfaces, on forwarding, on exit to interfaces, and on the way to (and from) applications on the router itself. Netfilter is configured by giving it a set of rules. A rule specifies the point at which to examine packets, the particular interface (where appropriate), a particular packet property to look for, and an action. An action can range from simply dropping matching packets to calling an arbitrary dynamically-loaded module for further processing. One of the modules available performs NAT.

Netfilter's NAT allows little control over where the packets it processes come from, or where they go after processing, since it is embedded at particular places in the IP processing path. Also, the way it finds returning packets is implicitly embedded in the code implementing Netfilter and IP, and not expressed in the configuration. This rules out a variety of advanced configurations. One example is embedding of NAT functions in an otherwise fully transparent Ethernet bridge; while Linux has bridging code, it cannot be combined with Linux's NAT code without kernel modifications. A second example is multiple fully independent NATs, which the Linux NAT code cannot support since it has just one connection state table. This might be useful for a load balancer, which might have multiple interfaces to back-end servers that shared the same private net address; a truly independent NAT per interface would allow returning packets to be associated with the correct NAT. Furthermore, independent tables, one per processor, can speed up NAT processing on a multiprocessor machine [4].

3 Design of a Click NAT

Here, we describe the NAT modules we built for Click, a modular networking system. (See Appendix A for an overview of Click.) Our components easily implement conventional NAT functions, like those described above, but avoid the limitations of conventional NATs. Multiple NAT tables can easily coexist in a single configuration, the placement of the NAT is flexible relative to other networking elements, and Click NAT easily supports unexpected uses, some of which we describe in Section 4.

Click divides the functions of a router into modular components called *elements*. In conventional operating systems, NAT hooks in to the IP processing path at fixed places. Such a design would be inappropriate for Click, where the processing path for a given router is essentially arbitrary. Furthermore, Click elements should generally be fine-grained, implementing limited functionality. Given these constraints, how should NAT functionality be divided into elements? How should the processing path—and other NAT components, such as application-level gateways—access rewriting functionality?

We started from a well-known principle: design around the data structures. In particular, the main NAT element, *IPRewriter*, corresponds to a single mapping table. To use multiple NAT tables in a configuration, you simply include multiple *IPRewriter* elements. The multiple entry and exit points provided by conventional NATs correspond to multiple input and output ports on the NAT elements. However, in Click, the user controls the semantics of input and output ports, which enables unexpected uses of the elements. The NAT elements export simple APIs to one another, facilitating the construction of ALGs and rewriter plugins.

Click currently has seven NAT-specific element classes: four mapping tables, a plugin that implements a particular mapping discipline (load-sharing NAT), and two application-level gateways. Figure 2 lists the element classes and their functions.

Click handles Section 2.2's NAT requirements and properties as follows:

- **Addressing units**. Different "rewriter" elements use different addressing units. *IPAddrRewriter*, which implements Basic NAT, uses the source address as its addressing unit. For *IPRewriter* and *TCPRewriter*, two NAPT elements, the addressing unit is a flow identifier; for *ICMPPingRewriter*, it is the triple of source address, destination address, and ICMP identifier. This list is clearly extensible; the user can simply write another element.

- **Mapping table**. Each "rewriter" element contains a single mapping table. When a packet arrives, the element performs the required mapping function: extracting the packet's

nection. Any NAPTs between server and client must be forewarned of this connection, since it initiates externally.

Element	Function	Supported packet types
IPRewriter	NAPT mapping table	TCP, UDP
IPAddrRewriter	Basic NAT mapping table	Any IP
TCPRewriter	NAPT mapping table for TCP with sequence number adjustment	TCP
ICMPPingRewriter	NAPT-like mapping table for ICMP pings	ICMP echoes, replies
RoundRobinIPMapper	mapping table plugin implementing load-sharing NAT	none
FTPPortMapper	application-level gateway for FTP	FTP control
ICMPRewriter	application-level gateway for ICMP errors	ICMP errors

FIGURE 2—Click NAT elements and their functions.

addressing unit, looking it up in the mapping table, rewriting the packet accordingly, and sending the packet along.

- **Fresh packet rules**. A flexible NAT should support general mechanisms for dividing packets into classes, and allow the user to associate a fresh packet rule with each class. Most NATs build in a single classification mechanism. Click NAT skirts this problem by avoiding the classification decision entirely.

Click elements, including the NAT elements, can have multiple *input ports* on which packets arrive, and multiple *output ports* on which packets are emitted. Classification elements in this scheme have one input port and multiple output ports: packets arriving on the single input are classified according to some criteria, then emitted on the corresponding outputs. Click comes with BPF-like classification elements, among many others, and users can add to the set however they like. Click's rewriter elements, then, implement no classification on their own; they support multiple input ports instead. Each input port corresponds to a packet class. The user divides packets into classes and routes them to the intended input ports. This open-ended design is maximally flexible.

Each input port is associated with a single fresh packet rule. Section 3.1 describes this further.

- **Garbage collection**. Click NAT elements feature user-specifiable timeouts for expiring unused mappings. The elements that handle TCP optionally time out closed connections earlier. Garbage collection proceeds incrementally, as new mappings appear. Helper elements can implement arbitrary expiration and/or rate-limiting behavior.

- **APIs for application-level gateways**. The Click NAT elements export simple APIs for looking up mappings, entering new mappings, and, in the case of *TCPRewriter*, changing sequence number adjustments on the fly. Application-level gateway elements like *FTPPortMapper* and *ICMPRewriter* use these internal APIs. User-level programs can also access them with `ioctl` commands.

- **Flexible placement**. Click's NAT elements can be placed anywhere in a configuration, and there can be as many of them as the user wants.

3.1 Fresh packet rules

Each input port on a rewriter element corresponds to a single fresh packet rule, which determines how fresh packets arriving on that input port are translated. (The rules are irrelevant for non-fresh packets: a packet whose corresponding mapping is already in the table is treated independently of the input port on which it arrived.) This section describes the fresh packet rules supported by Click. The *IPRewriter*, *IPAddrRewriter*, and *TCPRewriter* elements' configuration strings consist of a list of rules, one per input port.

Most fresh packet rules insert a pair of mappings into the rewriter element's mapping table. One of these mappings corresponds to the input packet, and applies to all packets with the same addressing unit. The other mapping applies to all reply packets—that is, packets that represent replies to the rewritten addressing unit. Say that the input packet had TCP flow ID (a_1, p_1, a_2, p_2),[*] and the fresh packet rule suggested the new flow ID (a_1', p_1', a_2', p_2'). (Most fresh packet rules will not change every address and port, however.) Then one of the installed mappings will map (a_1, p_1, a_2, p_2) to (a_1', p_1', a_2', p_2'), and the other, for reply packets, will map (a_2', p_2', a_1', p_1') to (a_2, p_2, a_1, p_1).

Fresh packet rules also indicate the output port on which the rewriter element should emit matching packets. Two output port numbers are generally listed, one for the original packet's addressing unit and one for reply packets. This turns rewriters into flow classifiers, since they remember the output port corresponding to a particular addressing unit. Several unexpected uses of *IPRewriter* take advantage of this facility.

The rule types are:

– '*drop*'. Fresh packets are dropped.

– '*passthru O*'. Fresh packets are passed through the translator element unchanged and emitted on output *O*. No new mappings are installed.

– '*keep* O_F O_R'. The rewriter installs a mapping that keeps the fresh packet unchanged, and the corresponding mapping

[*] Again, flow IDs are written (source address, source port, destination address, destination port).

for replies. The packet is emitted on output port number O_F; replies will be emitted on output port O_R. Thus, future packets of this connection will pass through the rewriter even if they arrive on an input port with, say, a '*drop*' rule.

- '*pattern* A_1 P_1 A_2 P_2 O_F O_R' (*IPRewriter* and *TCPRewriter* only). The first four parts of the pattern represent a new flow ID for the input packet: a new source address A_1, source port P_1, destination address A_2, and destination port P_2. The rewriter installs a mapping that changes the fresh packet's flow ID to the flow ID given by the pattern, and the corresponding mapping for replies. Packets similar to the fresh packet are emitted on output port O_F; replies are emitted on output port O_R.

  Any of the addresses and ports can be a dash '–', which means "leave unchanged". Thus, the pattern "*1.0.0.1 – 1.0.0.2 –*" will set the packet's source and destination addresses but leave the ports as they are. The source port specification also supports ranges "P_L–P_H", in which case the rewriter will choose a port between P_L and P_H. It will also ensure that any two active mappings created by this pattern have different source ports. The pattern '*1.0.0.1 1024–65535 1.0.0.2 80*', for example, sets every fresh packet's source address to 1.0.0.1, destination address to 1.0.0.2, and destination port to 80. The new source port, however, will differ for any two active sessions. Therefore, the new source port uniquely identifies an session, and every reply packet can be mapped back to a unique flow ID.

  Source ports are only unique within a single rewrite pattern. That is, different patterns may simultaneously allocate the same source port.

- '*pattern* A_1 A_2 O_F O_R' (*IPAddrRewriter* only). This limited version of *pattern* changes packets' source and destination addresses only. The A_1 argument may be a range of IP addresses, similar to P_1 above.

- '*pattern* name O_F O_R'. Named patterns, which are stored in a special *IPRewriterPatterns* element, can be shared by multiple rewriter elements, or by multiple input ports on a single rewriter element. This helps keep allocated addresses and/or source ports unique.

- 'elementname'. A fresh packet rule may consist of a single element name. This element must implement the *IPMapper* interface. When a fresh packet is encountered, the translator element will call one of that element's methods. The mapping helper may install new mappings using whatever criteria it likes. This makes the mapping mechanism arbitrarily extensible. Click comes with one *IPMapper* element, *RoundRobinIPMapper*, which assigns new flow IDs in a round-robin fashion among several choices.

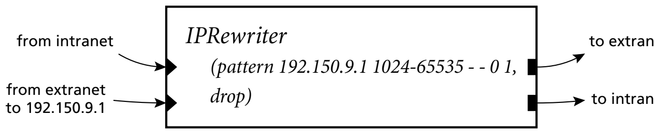
from intranet
from extranet to 192.150.9.1
IPRewriter
(pattern 192.150.9.1 1024-65535 - - 0 1, drop)
to extranet
to intranet

FIGURE 3—An *IPRewriter* element implementing NAPT.

  This implements load-sharing NAT. To implement a policy more complex than round-robin, the user need only write a new element.

3.2 *IPRewriter*

The rest of this section demonstrates several Click NAT elements: *IPRewriter* and *TCPRewriter*, *RoundRobinIPMapper*, and *FTPPortMapper*. We use conventional NAT applications as examples, which many currently available NATs can implement with more or less effort. For these applications, Click NAT's modularity makes the functions of individual elements, and the connections between them, clear and easy to manipulate. Section 4 presents some applications that are simple with Click NAT, but difficult to impossible with other NATs.

The *IPRewriter* and *TCPRewriter* translation elements implement network address port translation (NAPT). *IPRewriter* handles TCP and UDP; *TCPRewriter* is specialized for TCP, and can change packets' sequence and acknowledgment numbers as well as their addresses and ports.

Figure 3 shows an *IPRewriter* element set up for simple NAPT ("IP masquerading") with a single externally-visible IP address, 192.150.9.1. The element has two inputs and two outputs. The user arranges the configuration so that packets headed out of the internal network arrive on input port 0, and packets headed into the internal network arrive on input port 1. We'll describe the action of this element in detail.

The *IPRewriter*'s configuration string has two clauses, one per input port. First, fresh packets arriving on input port 0 represent new connections to the outside world. The NAPT should rewrite these packets to use its external IP address, allocating a new source port per connection and storing the mapping for later. A "*pattern*" rule, "*pattern 192.150.9.1 1024–65535 – –*", fits naturally. (This rule uses only non-reserved source ports.) Packets arriving on input port 1 represent new connections *from* the outside world and destined for the NAT's externally-visible address. Any such packets that are fresh should be dropped; thus, the "*drop*" rule. To figure out the semantics of the two output ports, we look at the mappings that might be installed by the rules. The "*drop*" rule never installs a mapping or emits a packet, so we can ignore it. The "*pattern*" rule installs two mappings: the forward mapping emits packets on output 0, the reply mapping on output 1. Since all packets arriving on input

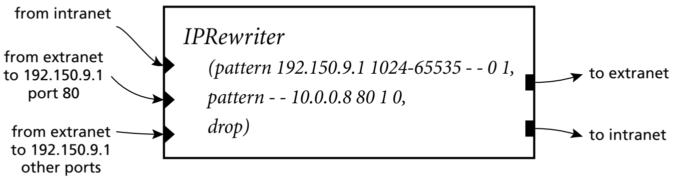
from intranet
from extranet to 192.150.9.1 port 80
from extranet to 192.150.9.1 other ports
IPRewriter
(pattern 192.150.9.1 1024-65535 - - 0 1,
pattern - - 10.0.0.8 80 1 0,
drop)
to extranet
to intranet

FIGURE 4—An *IPRewriter* element implementing NAPT, with redirection of port 80 to an internal Web server.

0 originated internally, we know that the forward mapping corresponds to packets originating internally (and headed outside), and the reverse mapping corresponds to packets originating externally (and heading inside). Thus, the *IPRewriter* sorts packets leaving the intranet onto output port 0, and packets entering the intranet onto output port 1.

Figure 4 extends this *IPRewriter* to redirect port 80 connections to an internal Web server at 10.0.0.8. All we do is add a new input port, for packets destined for the rewriter's port 80, and a corresponding rule, "*pattern – – 10.0.0.8 80*", which rewrites packets' destination addresses to that of the internal server. The new rule's output ports, "*1 0*", preserve the output port semantics from Figure 3. Some classification elements, not shown, select packets destined for port 80 and send them to the relevant input port. Extending Figure 3 into Figure 4 was easy. Given a new packet class, we added an input port (and the necessary classifiers) and a corresponding rule. Pleasantly, the configuration remains modular and readable.

3.3 *RoundRobinIPMapper*

Complex applications like load-sharing NAT, which distributes connections to a service among several machines, require more flexibility than the built-in fresh packet rules provide. Therefore, rewriters can delegate fresh packet handling to arbitrary "mapper plugin" elements. Those elements implement a C++ method, `get_map`, which a rewriter element calls when it encounters a fresh packet. Arguments specify the relevant rewriter and describe the fresh packet. The mapper plugin should choose a new mapping, install it into the rewriter, and return it.

RoundRobinIPMapper is one example mapper plugin. It has no inputs or outputs; packets don't pass through it.* Its configuration string is a list of fresh packet rules. On calls to `get_map`, it cycles through those rules in round-robin order, returning the first mapping it can allocate. Using *RoundRobinIPMapper*, connections to a single "virtual server" could be distributed round-robin to a set of real servers. Say that, starting with the NAT of Figure 4, we'd like to distribute connections to the internal Web server among machines 10.0.0.8, 10.0.0.9, and 10.0.0.10. First, we'd add a *RoundRobinIPMapper*:

```
rr_mapper :: RoundRobinIPMapper
   (pattern – – 10.0.0.8 80 1 0, pattern – – 10.0.0.9 80 1 0,
   pattern – – 10.0.0.10 80 1 0);
```

Then, we just replace the relevant line in the *IPRewriter*'s configuration string with a reference to *rr_mapper*:

```
IPRewriter(pattern 192.150.9.1 1024–65535 – – 0 1,
   rr_mapper, drop);
```

This simple design makes it easy to write and use new load balancers.

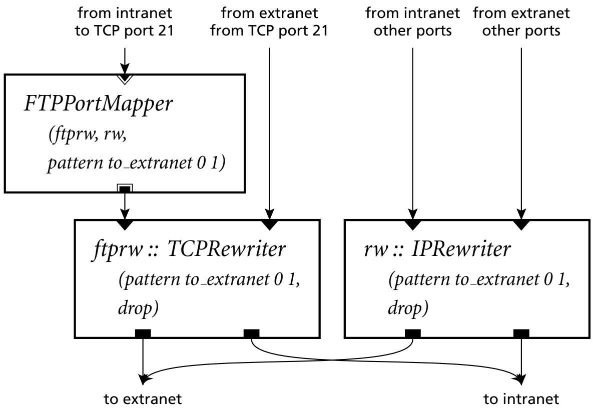
from intranet to TCP port 21
from extranet from TCP port 21
from intranet other ports
from extranet other ports
FTPPortMapper
(ftprw, rw,
pattern to_extranet 0 1)
ftprw :: TCPRewriter
(pattern to_extranet 0 1,
drop)
rw :: IPRewriter
(pattern to_extranet 0 1,
drop)
to extranet
to intranet

FIGURE 5—The *FTPPortMapper* ALG in context of a NAPT.

3.4 *FTPPortMapper*

To demonstrate how application-level gateways work in Click, we add an *FTPPortMapper* element to our NAPT, allowing FTP to pass through the NAT gateway. Recall that an FTP application-level gateway must (1) rewrite outgoing PORT commands embedded in the FTP control stream to use external addresses, (2) adjust sequence and acknowledgment numbers in the FTP control stream, and (3) install mappings for FTP data connections in the corresponding rewriter. The *FTPPortMapper* element takes three configuration arguments: the name of a *TCPRewriter* element handling FTP control streams (used for sequence number adjustment), the name of an *IPRewriter* or *TCPRewriter* element handling FTP data streams (used to install mappings; it may be equal to the control-stream element), and a fresh packet rule used to create and install mappings for data streams. Only FTP control packets leaving the intranet need pass through the *FTPPortMapper*. After leaving *FTPPortMapper*, packets must pass through the *TCPRewriter* mentioned in *FTPPortMapper*'s configuration string.†

Figure 5 shows how this might fit together. *FTPPortMapper* is placed in line with FTP control traffic, the only kind of

*In Click terminology, it is an *information element.*

†*FTPPortMapper* checks this property on initialization.

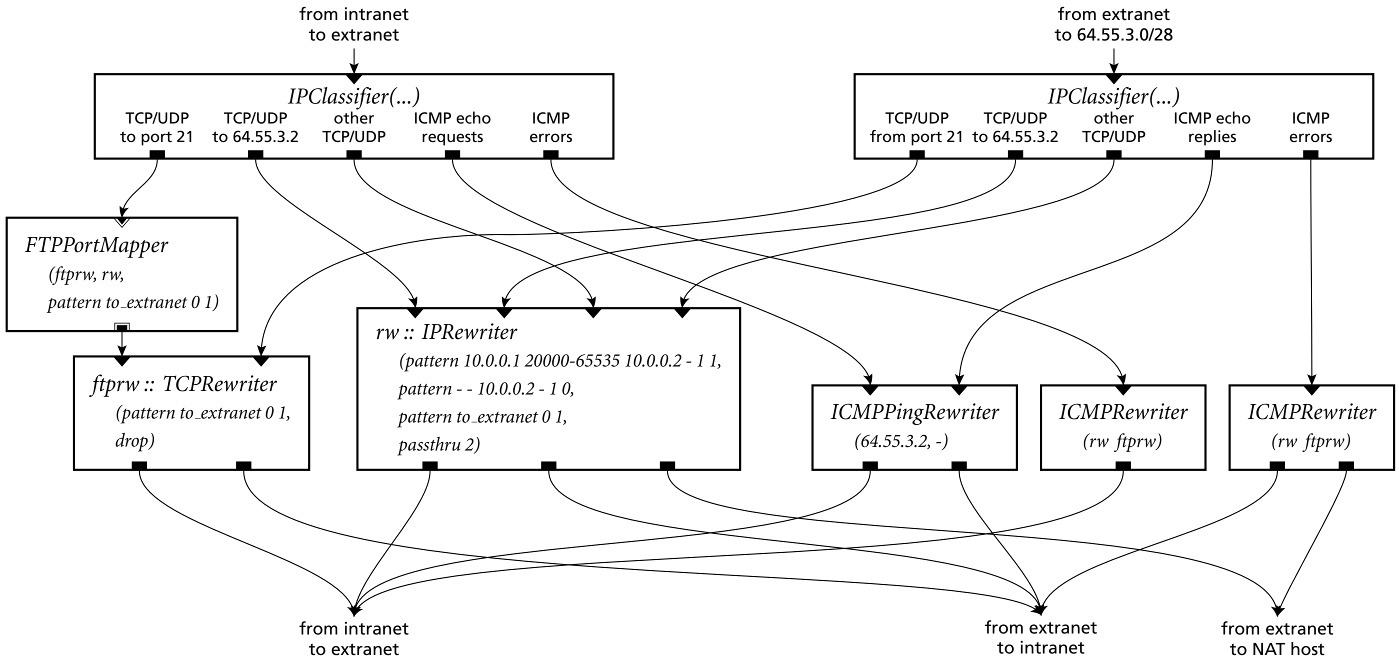

Figure 6—A complex firewalling NAPT configuration.

traffic it examines. Click's modularity clarifies the types of traffic that an application-level gateway affects. The named pattern, "*to_extranet*", prevents reuse of source ports among the different rewriter elements.

3.5 Discussion

Once a given rewrite pattern's source port range is exhausted, that pattern will drop new packets rather than reuse active source ports. As mappings are removed, of course, the corresponding source ports become available again.

Some care is required to ensure that different fresh packets are never mapped to the same new flow ID. For example, consider the following *IPRewriter* element:

IPRewriter(pattern 1.0.0.1 1024-65535 – – 0 1,
pattern 1.0.0.1 1024-65535 – – 0 1);

Clearly, these patterns conflict, which might make it impossible to determine the internal source address to which a reply packet should be sent. However, the rest of the configuration might ensure that packets arriving on input 0 had destination port 80, while packets arriving on input 1 had destination port 22. Then there would be no conflict between the patterns, because two new flow IDs created by the two patterns would never share the same destination port. Shared named patterns also prevent conflicts.

Click supports *hot swapping*, where a new configuration atomically takes the state of an old configuration on installation. The *IPRewriter* elements support hot swapping; their mapping tables need not be lost when configurations change.

4 Examples

The best way to demonstrate the flexibility and utility of the IP rewriting elements is through examples. We present first two complex, but conventional, configurations: a NAPT and a transparent traffic diverter. We close with several configurations easily supported by Click NAT that conventional NATs can't handle.

4.1 A complex NAPT

First, we present a more realistic and complex NAPT configuration. Versions of this configuration have been in daily use as a small startup company's Internet gateway for the past eighteen months. Its functional requirements include:

- Internal hosts have unlimited access to the extranet via TCP, FTP, UDP, and ICMP pings. ICMP errors, such as "port unreachable" messages, pass through to internal hosts.
- External hosts can initiate connections to two internal machines, a file server (external address 64.55.3.2, internal address 10.0.0.2) and the gateway itself (external address 64.55.3.1, internal address 10.0.0.1).
- Internal hosts can access the file server using either its internal address or its external address.

Figure 6 shows the corresponding configuration. We've seen its core before: the *FTPPortMapper*, *IPRewriter*, and *TCPRewriter* come from Figure 5. The *IPClassifier* elements,

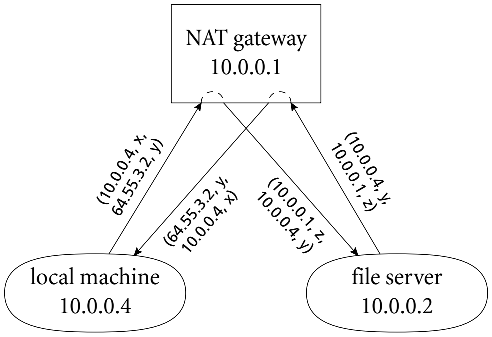
NAT gateway
10.0.0.1
(10.0.0.4, x, 64.55.3.2, y)
(64.55.3.2, y, 10.0.0.4, x)
(10.0.0.1, z, 10.0.0.4, y)
(10.0.0.4, y, 10.0.0.1, z)
local machine
10.0.0.4
file server
10.0.0.2

FIGURE 7—Rewriting packets headed for the file server's external address.

not shown earlier, divide packets into classes as appropriate for the configuration. The gateway's expanded requirements translate to configuration changes as follows:

- **ICMP echo support**. The *ICMPPingRewriter* element acts like an *IPRewriter* for ICMP echo requests and replies. Outgoing pings have their source addresses rewritten, to the externally visible address 64.55.3.2; incoming replies are rewritten correspondingly.
- **ICMP error support**. ICMP error packets, such as "port unreachable" or "network unreachable", must be rewritten if they are to cross a NAT. Each ICMP error packet contains a fragment of the offending packet's header, enough to extract a flow identifier. The *ICMPRewriter* elements use these flow identifiers to look up any corresponding mapping in the rewriter elements (*rw* and *ftprw*). If a mapping is found, they rewrite the ICMP error's destination address and enclosed packet header, and emit the rewritten error.
- **Access to the file server (64.55.3.2)**. External access to the file server uses the same technique as Figure 4. An input port on the *IPRewriter* accepts packets headed for the file server's external address, rewriting them to use the internal address with the rule "*pattern – – 10.0.0.2 – 1 0*". The new wrinkle is that we want to let *internal* hosts access the file server using its *external* address. We don't want to send those packets out into the extranet; they would cause ICMP redirects. Instead, we rewrite them to appear to come from the NAT gateway, using the rule "*pattern 10.0.0.1 20000-65535 10.0.0.2 – 1 1*". Both output ports are 1 because all communication is with the intranet. Figure 7 shows how this works for an example connection.
- **Access to the NAT gateway**. The "*passthru 2*" rule, which corresponds to TCP and UDP packets from the extranet, allows external connections to the NAT gateway. Packets without a mapping are emitted on the *IPRewriter*'s second output, from which they head to the NAT host's IP stack. The host will discard and/or log any inappropriate packets.

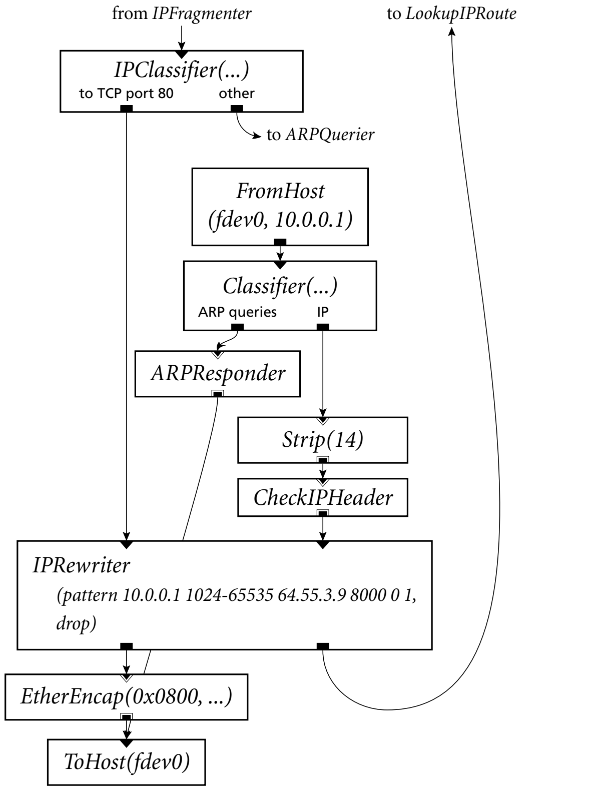

FIGURE 8—A transparent Web traffic diverter extension for the IP router configuration (Figure 12).

Figure 6's complexity derives from its requirements. Despite this complexity, Click's modularity makes it relatively easy to pick apart the configuration and see what it does, and to extend it if necessary.

4.2 Transparent traffic diverter

This section presents a NAT-based transparent traffic diverter suitable, for example, for turning ordinary proxies and servers into transparent proxies [7]. The diverter is meant to intercept all connections of a certain type, regardless of intended destination, and send them to a particular host and port. The connections arrive at that host looking as if they were originally intended to connect there. The program listening to the relevant port can accept the connections as if they were ordinary connections. When the program sends data on such connections, the diverter rewrites them to look as if they came from the host the connection was originally meant to connect to.

Figure 8 shows a Click configuration fragment that fits the diverter's *IPRewriter* into the larger IP router configuration of Figure 12 (in the appendix). Figure 8 catches outgoing traffic just before it reaches the outgoing interface's *ARPQuerier* and separates Web traffic from other traffic using an *IPClassifier*. Web traffic passes through an *IPRewriter* element, which diverts connections to port 8000 on the local

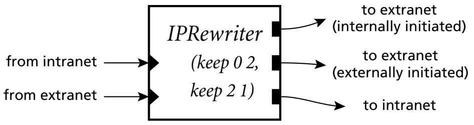

FIGURE 9—An *IPRewriter* for classifying packets into internally-initiated connections and externally-initiated connections.

machine (64.55.3.9). Packets' source addresses are changed to 10.0.0.1. This private address corresponds to a fake device, *fdev0*, created and installed in the host's device and routing tables by the *FromHost* element. The local machine's replies to the Web connections are then sent to 10.0.0.1. The fake device hands them to the Click configuration, where they are emitted by the *FromHost* element and eventually rewritten.

This diversion technique works for more than just HTTP traffic. For example, we used exactly this configuration fragment to build a transparent DNS cache that diverts DNS UDP packets to a host running an name server. The name server need not be modified at all as long as recursion is enabled.

4.3 Rate limiting by direction

The remaining examples present some uses of the IP rewriter elements that are difficult or impossible with conventional NAT. They are possible with Click because of the NAT elements' component nature—for example, multiple rewriters are possible in a single configuration—and because important properties of the NAT elements, such as the semantics associated with input and output ports, are determined by the user.

For our first example, we rate limit TCP connections differently depending on where they initiated. In particular, we might like to limit externally-initiated TCP connections to use at most half the available outgoing bandwidth. Currently, this kind of operation is approximated with per-port rate limits. However, some applications, such as peer-to-peer or network conferencing, may not use well-known ports, making the approximation inviable.

What's required here is a table that classifies connections into two categories, internally-initiated and externally-initiated. Click NAT easily handles this; see Figure 9. The two fresh packet rules, "*keep 0 2*" and "*keep 2 1*", do not change packets' addresses or TCP ports, but outbound packets due to internally-initiated and externally-initiated connections are sent to different output ports (0 and 1, respectively). The user could then hook a rate-limiting element, such as *RatedSplitter*, up to output 1. This use of NAT does no address translation whatsoever. Instead, it opportunistically uses *IPRewriter*'s mapping table to achieve an interesting effect. The repurposing succeeds because *IPRewriter* delegates port semantics to the user.

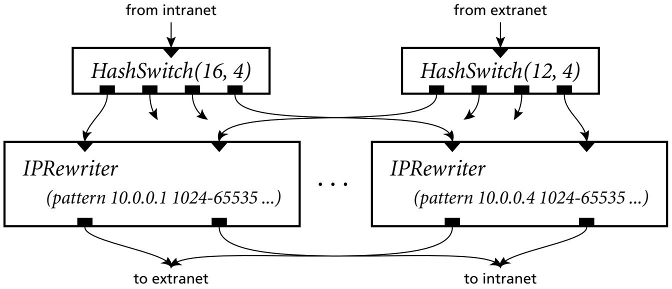

FIGURE 10—Multiple IP rewriters reduce SMP lock contention. The *HashSwitch* elements divide packets into classes based on external address; each class has a different *IPRewriter*.

By contrast, Linux 2.4's Netfilter does not currently support rate limiting by direction. Its connection state tables don't store accessible information about the direction in which connections were initiated. Of course, Linux could be extended, but Click's flexible elements support this use automatically.

4.4 Multiple NATs for SMP

Dividing expensive computations among multiple processors is a well-known technique for improving performance. Click can use multiple processors on SMP machines, but when doing so, mutable data structures, such as rewriter tables, should be touched by as few CPUs as possible [4]. Otherwise, the locking required to protect the data structures would drag down performance.

Conventional NATs allow at most one table per configuration, so locking overhead is unavoidable. Click, however, makes it easy to include multiple rewriters, as long as requests and replies for a given connection are always sent through the same rewriter. The configuration designer might choose, for example, to divide packets into classes based on the external IP address, which is visible as the destination address of outgoing packets and the source address of incoming packets. Different classes would then be routed to different rewriters, reducing the chance of lock contention; see Figure 10. Alternately, the user could try to allocate one *IPRewriter* per CPU or output interface.

4.5 Resilient overlay networks

To close this section, we sketch how another project, Resilient Overlay Networks [2], uses Click's IP rewriting elements. The RON project builds overlay networks designed to improve end-to-end reliability. One RON implementation is built on Click. In that implementation, RON clients encapsulate packets and send them via RON relays when their direct links to the Internet become unsuitable. RON clients use *IPRewriter*s to classify packets based on where their connections initiated. Connections initiated as part of the RON are routed according to RON's routing table; other packets use the conventional routing table, allowing simultaneous conventional and RON connections to the machine. These *IPRewriter*s simply maintain state about connections. RON relays, in contrast, use conventional NAPT to forward packets from clients to ordinary servers on the Internet. However, a RON relay might interact with many clients. It must remember the client corresponding to each mapping, so that it can correctly encapsulate reply packets from the Internet for the relevant client. This is easy; different clients get different input and output ports on the relay's *IPRewriter*. In both cases, Click NAT naturally handles extended requirements that the default Linux NAT framework cannot.

4.6 Summary

Click's NAT elements support conventional NAT operations, unusual uses of address translation like the traffic diverter, and unexpected uses like the rate limiter, equally well. The broad applicability of Click NAT's simple, well-specified components speaks well for the Click paradigm, and for modular networking systems in general.

5 Performance

IP rewriting in Click is sufficiently fast to make its performance impact negligible in the context of a larger router, firewall, or other packet processing configuration.

We evaluated the latency of an individual *IPRewriter* by using micro-benchmarks. Measurements were taken on a 700MHz Pentium III with 256MB of memory and a 256KB L2 cache, using Pentium performance counters. The test harness consists of a packet source element feeding fake UDP packets through an *IPRewriter* and into a packet sink.

In the first test, the packet source generates 100 identical packets. These packets create only one mapping, so the test measures the forwarding cost of the *IPRewriter* exclusive of the overhead of generating new mappings. *IPRewriter*'s median forwarding latency in this scenario is 393 cycles, or 561 ns, per packet.

In the second test, the packet source generates 100 packets with different UDP port numbers. This combines the overhead of packet forwarding with the overhead of generating new mappings, since each fresh packet adds two mapping entries to the *IPRewriter*'s hash table. The median latency for forwarding and generating a new mapping is 2338 cycles, or 3.34 μs, per packet. This is of course a worst-case value, since for normal traffic not every packet will result in a new mapping being created. For example, if every tenth packet starts a new flow, the average per-packet cost will be about 0.9 μs.

To put these numbers in context, a basic Click IP router (without NAT) on the same hardware takes about 2.8 μs of CPU time to forward a packet [12]. This includes Ethernet device interaction (with polling rather than interrupts) and all standard IP processing, such as IP checksum verification. The Linux 2.2.14 kernel IP forwarding code running on the same hardware, using Linux's interrupting drivers, takes about 12 μs of CPU time to forward a packet.

	MLFFR (packets/s)		
Configuration	**IP**	**NAT**	**Ratio**
Linux, interrupting	172,000	110,000	64%
Click, interrupting	222,000	210,000	94%
Click, polling	540,000	490,000	91%

FIGURE 11—Maximum loss-free packet forwarding rates for Click and Linux 2.2.18 configurations.

Figure 11 shows some macrobenchmarks of NAT forwarding rates. The router involved was a 1.2 GHz Pentium III, running Linux 2.2.18, with two Intel Pro/1000 F Server Adapter gigabit Ethernet cards in 64-bit 66 MHz PCI slots. The load was 10 flows of 64-byte UDP packets, in one direction only; each flow had a unique and unchanging address/port-number combination. The numbers in the table are the maximum input rates that the router could forward with no losses. The lines marked "interrupting" involve an interrupting driver. The line marked "polling" was taken with a driver modified to allow Click to poll for incoming packets, thus avoiding all interrupts. The IP column refers to standard IP forwarding; for Click, this means the configuration in Figure 12. The NAT column refers to a simple NAPT configuration. For Click, this is *IPRewriter*; for Linux, it was produced by the command `ipchains -A forward -s 1.0.0.0/8 -j MASQ`.

Figure 11 should not be interpreted as a direct comparison of Click and Linux. For example, Linux's forwarding table lookup algorithms are more complex than Click's, but scale better to large numbers of routes. However, the figure shows that Click's NAT implementation adds only modestly to the cost of basic IP forwarding, and that Click's modular separation of NAT from other forwarding tasks does not prevent it from performing as well as a more conventional architecture.

6 Related Work

The first documented IP network address translator performed address-based NAT only [9]. Network Address Port Translation, or NAPT, and the use of NAT for load balancing appeared later [16, 17].

RFC 2663 lays out consistent terminology for NAT variants [17]. Click NAT elements can perform Basic NAT with *IPAddrRewriter*, Network Address Port Translation with *IPRewriter*, Load Sharing NAT with *IPRewriter* plus an *IPMap-*

per element, Two-Way NAT with *IPRewriter* and a DNS proxy, Twice-NAT with two *IPRewriter*s in different realms, and Multihomed NAT.

Hasenstein describes a wide variety of network address translation configurations in the context of a system for NAT in Linux [11]. All of his configurations may be easily implemented in Click.

Cisco IOS's NAT implementation [5] supports static and dynamic address-based NAT, NAPT, round-robin load sharing NAT, and combinations thereof. Interfaces are divided into two classes, "inside" and "outside". The translations applied to a particular packet depend on the class of interface on which it was received, and, optionally, on its source address, destination address, protocol, or port number. Other arrangements, such as more than two classes of interface or other load sharing arrangements, are difficult or impossible to achieve.

Similarly, while the NAT implementations shipped with desktop operating systems—Linux's ipchains and ipnat [11] and Netfilter [1], BSD's IP Filter [15], and Windows 2000's NAT—are flexible to different degrees, none of them support multiple NAT components in a single configuration, or allow fully flexible control over NAT placement relative to other forwarding tasks.

Cohen et al. [7, 8] present a configurable tool for re-mapping packet addresses and port numbers. It consists of a kernel module that implements re-mapping with a fixed table, and applications that add new mappings when they observe packets from as-yet unmapped flows. While the system can perform a wide range of NAT functions, it is embedded inside a fixed router configuration; unlike Click's NAT tools, the way it interacts with other forwarding functions cannot be changed.

7 Conclusion

We have presented a flexible set of components for network address translation in Click, a modular networking system. These components implement only the core functionality required of any network address translator—namely, changing IP packet headers and finding mappings corresponding to input packets. They leave other functions, such as determining which packets should be subject to translation, to other parts of the configuration. This makes configurations involving address translation flexible and understandable. NAT elements can be placed in a configuration exactly where they are required; packets meant for translation can be selected in arbitrary ways; the mechanism for choosing a translation for a new packet is completely extensible; and multiple NAT elements can coexist in a single configuration. The IP rewriting components are made more useful and general by the modular networking system of which they are a part. We demonstrated the practical usefulness of this system with real configurations, including an IP router with port translation, a transparent traffic diverter, and several configurations impossible to achieve with conventional NATs, and showed that the IP rewriting elements have acceptable performance cost.

The components described in this paper, and several example configurations, are freely available on line at `http://www.pdos.lcs.mit.edu/click/`.

Acknowledgments

We thank Paul Hsiao, Sulu Mamdani, Dean Bogdanovic, Prem Gopalan, and Mazu Networks, Inc. for support of this project, Benjie Chen for his initial work on the configuration described in Section 4.1, Alex Yip for the RON example, and Frans Kaashoek and the members of MIT LCS's Parallel and Distributed Operating Systems group for supporting Click.

References

[1] The netfilter/iptables project. Technical report. `http://www.netfilter.org/`, as of January 2002.

[2] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, October 2001.

[3] F. Baker. Requirements for IP Version 4 routers. RFC 1812, Internet Engineering Task Force, June 1995. `ftp://ftp.ietf.org/rfc/rfc1812.txt`.

[4] Benjie Chen and Robert Morris. Flexible control of parallelism in a multiprocessor PC router. In *Proc. 2001 USENIX Annual Technical Conference (USENIX '01)*, June 2001.

[5] Cisco Systems. Cisco IOS Network Address Translation (NAT). Technical report, 2001. `http://www.cisco.com/warp/public/cc/pd/iosw/ioft/ionetn/prodlit/1195_pp.htm`, as of February 2002.

[6] Click Project. The Click modular router: Documentation. Technical report, 2002. `http://www.pdos.lcs.mit.edu/click/doc/`, as of February 2002.

[7] A. Cohen, S. Rangarajan, and N. Singh. Supporting transparent caching with standard proxy caches. In *Proceedings of the 4th International Web Caching Workshop*, 1999.

[8] Ariel Cohen and Sampath Rangarajan. A programming interface for supporting IP traffic processing. In *Proc. of IWAN '99: Active Networks, First International Working Conference*, number 1653 in Lecture Notes in Computer Science, pages 132–143, June 1999.

[9] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631, Internet Engineering Task Force, May 1994. `ftp://ftp.ietf.org/rfc/rfc1631.txt`.

[10] T. Hain. Architectural implications of NAT. RFC 2993, Internet Engineering Task Force, November 2000. `ftp://ftp.ietf.org/rfc/rfc2993.txt`.

[11] Michael Hasenstein. IP network address translation. Diplomarbeit, Technische Universität Chemnitz, Chemnitz, Germany, 1997. Available online at `http://www.suse.de/~mha/linux-ip-nat/diplom/nat.html`, as of February 2002.

[12] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(4), November 2000.

[13] E. Nordmark. Stateless IP/ICMP Translation algorithm (SIIT). RFC 2765, Internet Engineering Task Force, February 2000. `ftp://ftp.ietf.org/rfc/rfc2765.txt`.

[14] J. Postel and J. Reynolds. File Transfer Protocol (FTP). RFC 959, Internet Engineering Task Force, October 1985. `ftp://ftp.ietf.org/rfc/rfc0959.txt`.

[15] Darren Reed. IP Filter TCP/IP packet filtering package. Technical report, 2002. `http://coombs.anu.edu.au/~avalon/`, as of February 2002.

[16] P. Srisuresh and D. Gan. Load sharing using IP Network Address Translation (LSNAT). RFC 2391, Internet Engineering Task Force, August 1998. `ftp://ftp.ietf.org/rfc/rfc2391.txt`.

[17] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) terminology and considerations. RFC 2663, Internet Engineering Task Force, August 1999. `ftp://ftp.ietf.org/rfc/rfc2663.txt`.

[18] G. Tsirtsis and P. Srisuresh. Network Address Translation—Protocol Translation (NAT-PT). RFC 2766, Internet Engineering Task Force, February 2000. `ftp://ftp.ietf.org/rfc/rfc2766.txt`.

A Click

Click routers are built from components called *elements*. Elements are modules that process packets; they control every aspect of router packet processing. Router configurations are directed graphs with elements as the vertices. The edges, called *connections*, represent possible paths that packets may travel. Each element belongs to an *element class*, which determines the code executed when the element processes a packet. Each element also has an optional *configuration string*, which element classes can use to select behavior more precisely. For example, the *Tee* element class duplicates packets; a *Tee* element's configuration string, an integer, says how many copies to make. Inside a running router, elements are represented as C++ objects and connections are pointers to elements. A packet transfer from one element to the next is implemented with a single virtual function call.

Elements also have *input* and *output ports*, which serve as the endpoints for packet transfers. Every connection leads from an output port on one element to an input port on another. An element can have zero or more of each kind of port. Different ports can have different semantics; for example, the second output port is often reserved for erroneous packets.

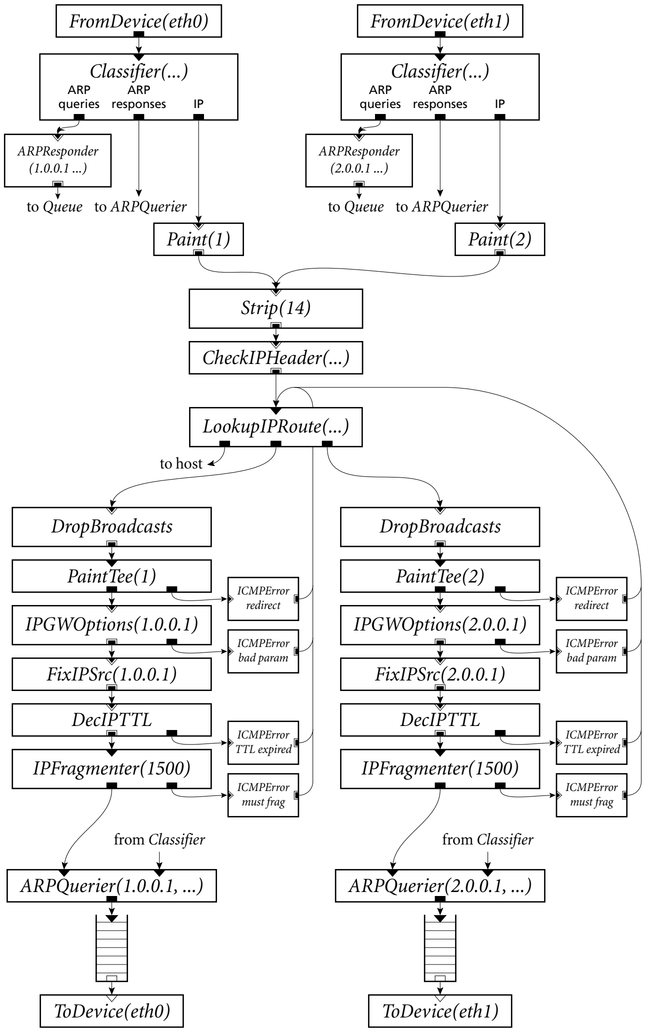

FIGURE 12—An IP router configuration.

Every queue in a Click configuration is explicit. Thus, a configuration designer can control where queueing takes place by deciding where to place *Queue* elements.

Click uses a simple, declarative language to describe router configurations. The language specifies how elements should be connected together. To configure a router, the user passes a Click-language file to the system. The system parses the file, creates the corresponding router, tries to initialize it, and, if initialization is successful, installs it and starts routing packets with it.

Figure 12 shows a basic 2-interface IP router configuration, the starting point for some of the configurations described in the body of the paper. This configuration implements all required IP forwarding functionality [3]; see [12] for a description of how it works.

Click router configurations run in a downloadable Linux kernel module, at user level, or in a FreeBSD kernel.